

OBJEKTIF :

1. Mahasiswa Menguasai Pemograman Pada Shell Script.
 2. Mahasiswa Mengetahui Variabel, Struktur Percabangan, dan Perulangan Pada Linux
-

2.1 DEFINISI SHELL SCRIPT

Shell script adalah *file* yang berisikan *command* yang dapat di eksekusi yang disimpan dalam bentuk text *file*. Ketika *file* dijalankan, setiap *command* akan di eksekusi. *Shell script* memiliki akses ke semua *command* yang ada di *shell*, termasuk *logic*. Oleh karena itu, *script* dapat menguji keberadaan *file* atau mencari *output* tertentu dan mengubah perilakunya. Anda dapat membuat *script* untuk mengotomatisasikan bagian yang berulang (repetitif) dari pekerjaan Anda, sehingga lebih menghemat waktu dan sudah dijamin konsisten setiap kali *script* digunakan. Misalnya, jika Anda menjalankan 5 perintah yang sama setiap hari, Anda dapat mengubah *command* itu ke dalam *shell script* yang dapat mengurangi pekerjaan Anda menjadi satu *command* saja.

Sebuah *script* bisa sesederhana dengan satu *command*, seperti:

```
echo "Hello, World!"
```

Script `test.sh` terdiri dari hanya satu baris yang akan mencetak *string* `Hello, World!` pada konsol.

Menjalankan sebuah *script* dapat dilakukan dengan meneruskannya sebagai *argument* ke dalam *shell* atau menjalankannya langsung:

```
sysadmin@localhost:~$ sh test.sh
```

```
Hello, World!
sysadmin@localhost:~$ ./test.sh
-bash: ./test.sh: Permission denied
sysadmin@localhost:~$ chmod +x ./test.sh
sysadmin@localhost:~$ ./test.sh
Hello, World!
```

Pada contoh diatas, *script* dijalankan sebagai *argument* ke dalam *shell*. Selanjutnya *script* dijalankan secara langsung dari *shell*. Jarang memiliki *current directory* di dalam pencarian *biner* `$PATH` sehingga namanya diawali dengan `./` untuk menunjukkan bahwa itu harus dijalankan dari direktori saat ini (*current directory*).

`Error permission denied` berarti menunjukan bahwa *script* belum ditandai sebagai *script* yang dapat di eksekusi. Untuk itu perlu digunakan `chmod`. `chmod` digunakan untuk merubah *permission* dari sebuah *file*, untuk lebih jelasnya akan dibahas pada bab lainnya.

Ada berbagai macam *shell* dengan sintaksnya masing-masing. Oleh karena itu, *script* yang lebih rumit akan menunjukkan *shell* tertentu dengan menentukan *path* absolut ke interpreter pada baris pertama, diawali dengan `#!` seperti:

```
#!/bin/sh
echo "Hello, World!"
```

atau

```
#!/bin/bash
echo "Hello, World!"
```

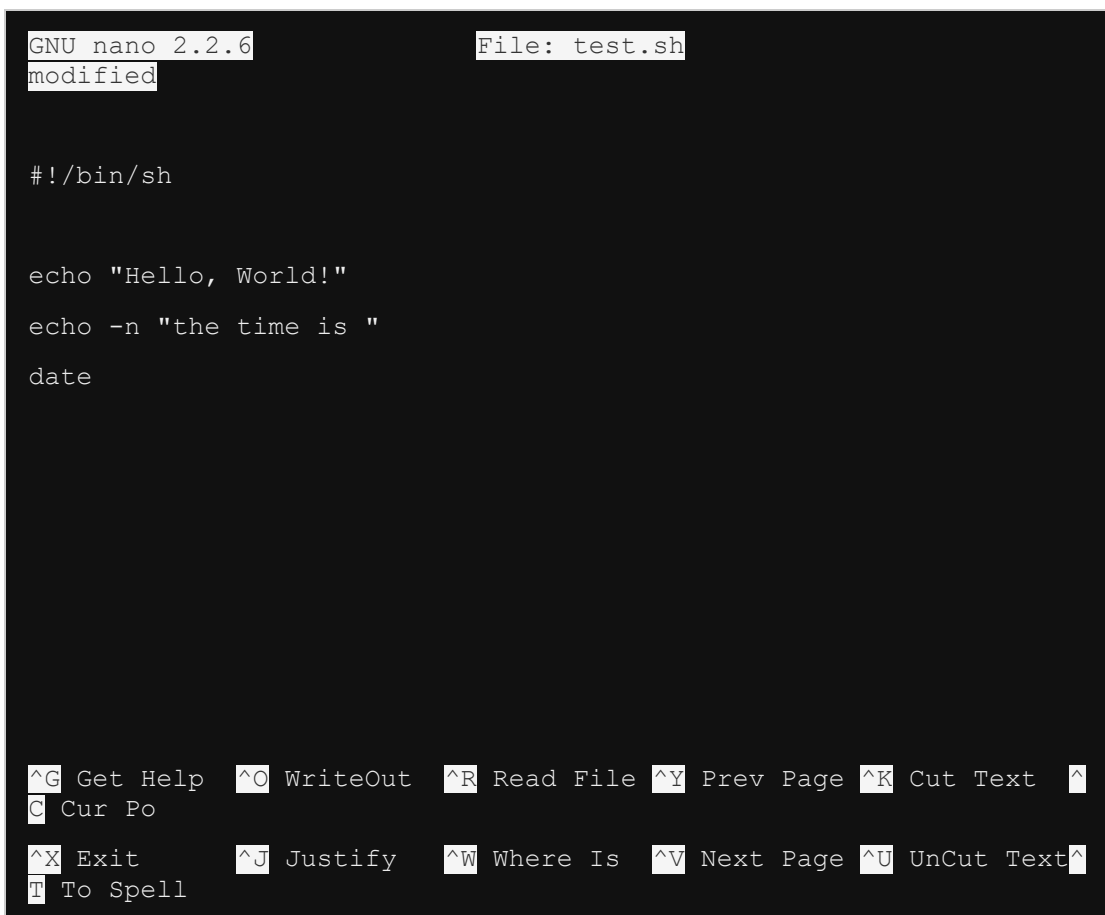
Karakter `#!` merupakan karakter *hash* dan *bang* yang disebut sebagai "*shebang*" ketika digunakan di bagian awal *script*. *Shebang* (atau *crunchbang*) digunakan untuk *shell script* tradisional dan untuk bahasa berbasis teks lainnya seperti Perl, Ruby, dan Python. *File* teks apa pun yang ditandai sebagai *executable* akan dijalankan dibawah interpreter yang ditentukan di baris pertama selama

script dijalankan secara langsung. Jika *script* dipanggil secara langsung sebagai argumen, seperti `sh script` atau `bash script`, *shell* tersebut akan menggunakan apa pun yang ada di baris shebang.

MENGEDIT SHELL SCRIPT

UNIX memiliki berbagai macam text editor. Dua diantaranya yang sering digunakan adalah GNU nano editor yang merupakan editor yang simple dan cocok untuk mengedit file teks berukuran kecil. Dan Visual Editor, vi, atau versi terbarunya, VI Improved (vim), yang merupakan editor yang powerful. Kita hanya akan fokus membahas mengenai nano editor.

Ketikkan `nano test.sh` dan Anda akan melihat tampilan seperti dibawah:



```
GNU nano 2.2.6 File: test.sh
modified

#!/bin/sh

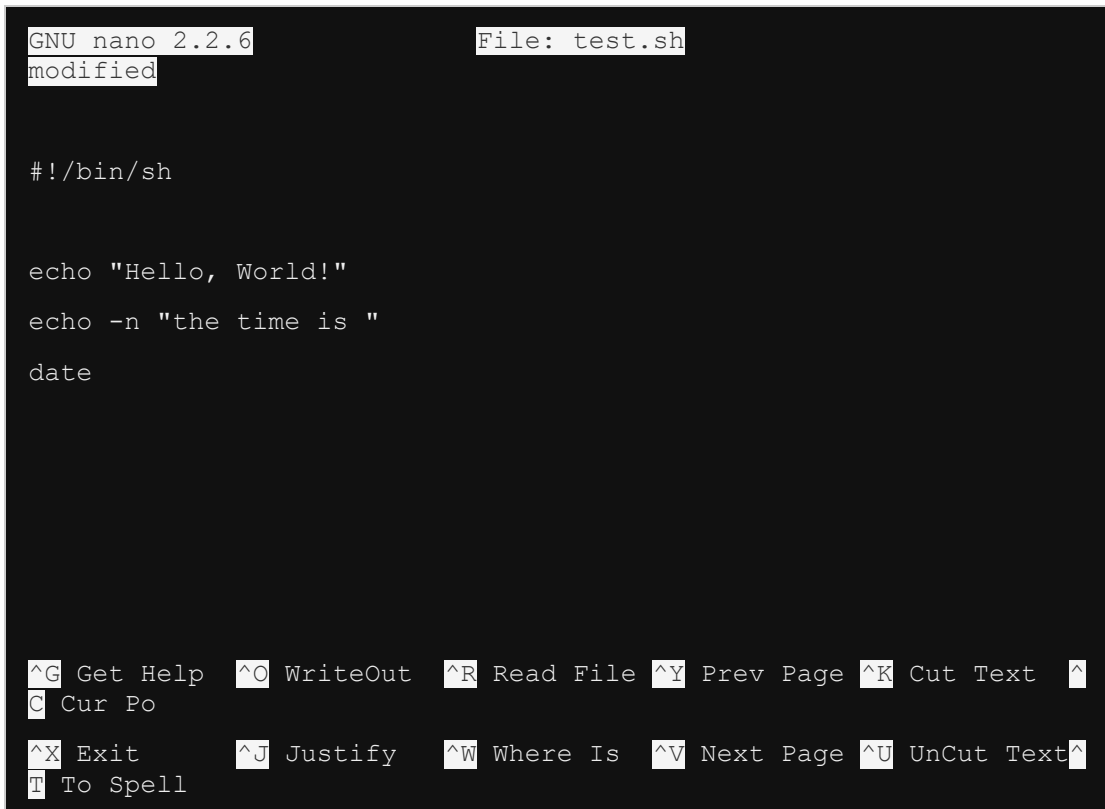
echo "Hello, World!"
echo -n "the time is "
date

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text  ^
^C Cur Po
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text^
^T To Spell
```

Nano *editor* memiliki beberapa fitur. Anda dapat dengan mudah mengetik dengan keyboard, dengan menggunakan **arrow keys** untuk berpindah-pindah dan

tombol ***delete/backspace*** untuk menghapus teks. Jika Anda langsung menggunakan mesin Linux, bukan dengan dihubungkan dengan jaringan, Anda juga dapat menggunakan *mouse* untuk memindahkan kursor dan menyorot teks.

Untuk membiasakan diri dengan *editor*, mulailah mengetik *script shell* sederhana di dalam **nano**:



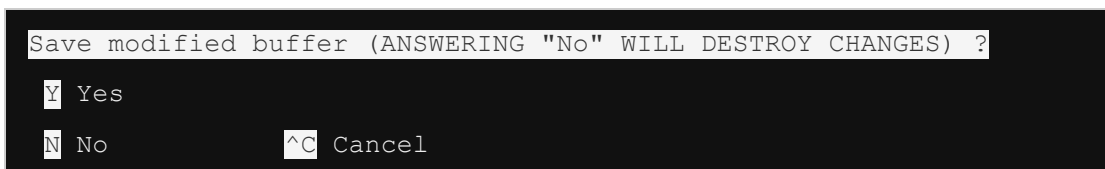
```
GNU nano 2.2.6 File: test.sh
modified

#!/bin/sh

echo "Hello, World!"
echo -n "the time is "
date

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text  ^
C Cur Po
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text^
T To Spell
```

Dibawah *editor* terdapat beberapa *command* yang dapat membantu Anda. Seperti *command* pada kiri bawah **^X Exit** yang berarti Anda harus menekan tombol **Ctrl** dan **X** secara bersamaan. Maka tampilan akan berubah seperti:



```
Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?
Y Yes
N No          ^C Cancel
```

Disini Anda dapat keluar dari program tanpa menyimpan dengan menekan tombol **N** pada keyboard, atau simpan terlebih dahulu dengan menekan **Y**. Anda dapat menekan **Enter** setelahnya untuk menyimpan dan keluar.

Anda akan Kembali ke *shell* setelah menyimpan. Kembali ke *editor* Kali ini tekan **Ctrl** dan **O** bersamaan untuk menyimpan pekerjaan Anda tanpa keluar dari *editor*. Arahkan kursor ke baris "The time is". Tekan tombol **Ctrl** dan **K** dua kali untuk *cut* dua baris terakhir ke dalam *copy buffer*. Arahkan kursor ke baris lain dan tekan **Ctrl** dan **U** untuk *paste*. Hal ini membuat *script* menampilkan *current time* sebelum Hello, World!.

Perintah pada nano editor lainnya yang harus diketahui:

Command	Deskripsi
Ctrl + W	Cari dokumen
Ctrl + W , lalu Ctrl + R	Cari dan replace
Ctrl + G	Bantuan
Ctrl + Y/V	page up / down
Ctrl + C	Menampilkan current position pada file dan ukuran file

2.2 Variabel Pada Linux

Variabel adalah bagian penting dari bahasa pemrograman. Contoh penggunaan sederhana dari variabel adalah:

```
#!/bin/bash

ANIMAL="penguin"

echo "My favorite animal is a $ANIMAL"
```

Setelah baris *shebang* adalah contoh penggunaan dari variabel. Nama dari variabel tersebut adalah `ANIMAL` yang menetapkan *string* penguin dengan

tanda sama dengan. Variabel itu seperti sebuah kotak dimana Anda dapat menyimpan berbagai macam barang. Setelah mengeksekusi baris ini, kotak bernama `ANIMAL` berisi kata `penguin`.

Perlu diingat bahwa tidak ada spasi diantara nama variabel, tanda sama dengan, dan item yang akan ditetapkan sebagai variabel. Jika Anda menggunakan spasi, maka akan terjadi *error* berupa “`command not found`”. Tidak diharuskan untuk membuat nama variabel dengan huruf kapital, tetapi penggunaan huruf kapital pada variabel dapat berguna sebagai pemisah antara variabel dari perintah yang akan dijalankan.

Selanjutnya, *script* mencetak *string* ke *console*. String tersebut berisikan nama dari variabel yang ditandai dengan tanda dollar (\$). Ketika interpreter bertemu dengan tanda dollar, maka interpreter akan mengganti isi dari variabel tersebut, yang disebut sebagai *interpolation*. Maka *output* dari *script* tersebut adalah `My favorite animal is a penguin`.

Jadi, perlu diingat: untuk menetapkan ke sebuah variabel, cukup gunakan nama dari variabel. Untuk mengakses isi dari variabel, maka gunakan awalan tanda dollar (\$). Berikut kita tunjukkan sebuah variabel diisi dari variabel lainnya:

```
#!/bin/bash

ANIMAL=penguin
SOMETHING=$ANIMAL
echo "My favorite animal is a $SOMETHING"
```

`ANIMAL` berisikan *string* `penguin` (tidak ada spasi, dan pada contoh ini ditunjukkan penggunaan sintaks tanpa menggunakan tanda kutip). Variabel `SOMETHING` kemudian diisi oleh `ANIMAL` (karena `ANIMAL` memiliki awalan tanda dollar).

Cara lain untuk menetapkan variabel adalah dengan menggunakan *output* dari perintah lain sebagai isi variabel dengan menyertakan *command* di dalam tanda *back ticks* (```), seperti:

```
#!/bin/bash
CURRENT_DIRECTORY=`pwd`
echo "You are in $CURRENT_DIRECTORY"
```

Back ticks tersebut sering digunakan untuk memproses teks. Anda dapat mengambil teks dari satu variabel atau *file input* dan meneruskannya melalui perintah lain seperti `sed` atau `awk` untuk mengambil bagian tertentu dan menyimpannya ke dalam variabel. Perintah `sed` digunakan untuk mengedit streams (STDIN) dan perintah `awk` biasa digunakan untuk *scripting*. Kedua perintah tersebut tidak akan dibahas pada modul ini.

Memungkinkan untuk mendapatkan *input* dari pengguna *script* dan menetapkan *input* tersebut ke dalam variabel dengan menggunakan perintah `read`.

```
#!/bin/bash

echo -n "What is your name? "
read NAME
echo "Hello $NAME!"
```

Perintah `read` dapat menerima *string* langsung dari keyboard atau sebagai bagian dari perintah *redirection* yang sudah dipelajari pada sub bab sebelumnya.

Terdapat beberapa variabel khusus selain yang sudah ada. Anda dapat menambahkan argumen ke dalam *script* Anda seperti:

```
#!/bin/bash
echo "Hello $1"
```

Tanda dollar (\$) diikuti dengan angka N sesuai dengan argumen ke-N yang diteruskan ke *script*. Jika Anda menjalankan contoh diatas dengan `./test.sh`

`World` maka hasil *output* nya akan menjadi Hello World. Variabel `$0` berisi nama dari *script* itu sendiri.

Setelah program dijalankan, baik itu dengan cara biner maupun dengan *script*, itu akan mengembalikan sebuah *exit code* yang berupa *integer* antara 0 hingga 255. Anda dapat mencobanya dengan variabel `$?` untuk melihat apakah perintah sebelumnya sudah berhasil dijalankan.

```
sysadmin@localhost:~$ grep -q root /etc/passwd
sysadmin@localhost:~$ echo $?
0
sysadmin@localhost:~$ grep -q slartibartfast /etc/passwd
sysadmin@localhost:~$ echo $?
1
```

Perintah `grep` digunakan untuk mencari *string* dari sebuah *file* dengan flag `-q`, yang berarti “*quiet*”. Perintah `grep`, ketika dijalankan dalam *mode quiet*, mengembalikan nilai 0 jika string ditemukan. Dan mengembalikan nilai 1 jika *string* tidak ditemukan. Informasi ini dapat digunakan dalam percabangan untuk melakukan suatu tindakan berdasarkan *output* atau perintah lainnya.

Demikian juga Anda dapat mengatur *exit code* dari *script* Anda sendiri dengan perintah `exit` :

```
#!/bin/bash
# Something bad happened!
exit 1
```

Contoh diatas menunjukkan *comment* dengan menggunakan tanda `#`. Apa pun setelah tanda pagar akan diabaikan dan tidak dieksekusi, yang mana dapat digunakan untuk membantu *programmer* untuk membuat catatan. `exit 1` mengembalikan nilai *exit code* 1. Jika Anda menjalankan *script* ini dari CLI dan menuliskan `echo $?` maka akan menghasilkan *output* 1.

Exit code 0 berarti “semuanya OK”. Jika *exit code* yang keluar lebih besar dari 0 berarti terdapat *error*. Pada contoh di atas dapat dilihat bahwa `grep` menggunakan 1 yang berarti *string* tidak ditemukan.

2.3 STRUKTUR PERCABANGAN PADA LINUX

Sekarang Anda dapat mengatur variabel, sekarang waktunya membuat skrip dengan melakukan fungsi yang berbeda berdasarkan tes, yang disebut *branching* atau percabangan. *If statement* adalah operator dasar untuk mengimplementasikan *branching*.

Bentuk umum dari *if statement* adalah sebagai berikut:

```
if somecommand; then
    # do this if somecommand has an exit code of 0
fi
```

Contoh berikutnya akan menjalankan "somecommand" (sebenarnya, semuanya hingga titik koma) dan jika *exit code*-nya adalah 0 lalu isinya sampai penutup `fi` akan dijalankan. Menggunakan apa yang Anda tahu tentang `grep`, sekarang Anda dapat menulis skrip yang dapat melakukan berbagai macam hal berdasarkan keberadaan string di *password file*:

```
#!/bin/bash

if grep -q root /etc/passwd; then
    echo root is in the password file
else
    echo root is missing from the password file
fi
```

Dari contoh sebelumnya, Anda mungkin ingat bahwa *exit code* dari `grep` adalah 0 jika string ditemukan. Contoh di atas menggunakan ini dalam satu baris untuk mencetak pesan jika `root` ada di *password file* atau mencetak pesan yang berbeda jika tidak. Perbedaannya di sini adalah alih-alih `fi` untuk menutup blok

`if`, ada juga yang namanya `else`. Ini memungkinkan Anda melakukan satu tindakan jika kondisinya benar, dan melakukan tindakan lainnya jika kondisinya salah. Blok `else` masih harus ditutup lagi dengan `fi`.

Tugas umum lainnya adalah mencari keberadaan dari suatu file atau direktori dan membandingkan string dan angka. Anda mungkin perlu menginisialisasi file log jika tidak ada, atau membandingkan jumlah baris dalam file pada saat terakhir kali Anda menjalankannya. Perintah `if` jelas perintah yang membantu di sini, tapi perintah apa yang Anda gunakan untuk membuat perbandingan?

Perintah `test` memberi kemudahan untuk perbandingan dan operator uji file. Sebagai contoh:

Perintah	Deskripsi
<code>test -f / dev / ttyS0</code>	0 jika file tersebut ada
<code>test ! -f / dev / ttyS0</code>	0 jika file tidak ada
<code>test -d / tmp</code>	0 jika direktori tersebut ada
<code>test -x `which ls`</code>	gantikan lokasi <code>ls</code> lalu <code>test</code> jika pengguna dapat mengeksekusi
<code>test 1 -eq 1</code>	0 jika perbandingan numerik berhasil
<code>test ! 1 -eq 1</code>	NOT – 0 jika perbandingan gagal
<code>test 1 -ne 1</code>	Lebih mudah, test untuk pertidaksamaan numerik

Perintah	Deskripsi
<code>test "a" = "a"</code>	0 jika perbandingan string berhasil
<code>test "a" != "a"</code>	0 jika string berbeda
<code>test 1 -eq 1 -o 2 -eq 2</code>	-o adalah OR: bisa jadi sama
<code>test 1 -eq 1 -a 2 -eq 2</code>	-a adalah AND: keduanya harus sama

Penting untuk diingat bahwa `test` terlihat berbeda pada integer dan string. `01` dan `1` sama dalam perbandingan numerik, tetapi berbeda pada perbandingan string. Anda harus berhati-hati mengingat jenis input yang Anda inginkan.

Ada lebih banyak perbandingan, seperti `-gt` untuk lebih besar dari, cara untuk mengetes apakah file itu lebih baru dari yang lain, dan banyak lagi. Lihat halaman `test man` untuk penjelasan lebih detail.

Perintah `test` cukup *verbose* untuk perintah yang sering digunakan, jadi terdapat sebutan untuk itu yang disebut `[` (tanda kurung siku kiri). Jika Anda menyertakan *conditionals* di dalam tanda kurung siku, itu sama dengan menjalankan `test`. Jadi, bentuk kedua *statement* di bawah ini adalah sama.

```
if test -f /tmp/foo; then
if [ -f /tmp/foo]; then
```

Meskipun bentuk terakhir paling sering digunakan, penting untuk diingat bahwa tanda kurung siku merupakan perintahnya sendiri yang beroperasi sama pada `test` kecuali memerlukan tanda kurung siku tutup.

if statement memiliki bentuk akhir yang memungkinkan Anda melakukan banyak perbandingan sekaligus menggunakan *elif* (kependekan dari *else if*).

```
#!/bin/bash

if [ "$1" = "hello" ]; then
    echo "hello yourself"
elif [ "$1" = "goodbye" ]; then
    echo "nice to have met you"
    echo "I hope to see you again"
else
    echo "I didn't understand that"
fi
```

Kode di atas membandingkan argumen pertama pada skrip. Jika itu *hello*, blok pertama akan dieksekusi. Jika tidak, skrip akan memeriksa untuk melihat apakah argumen tersebut *goodbye* dan *echos* (mencetak) pesan yang berbeda jika itu benar. Jika tidak, pesan ketiga akan dicetak. Perhatikan bahwa variabel *\$1* dikutip dan operator perbandingan string digunakan dari pada menggunakan operator perbandingan numerik (*-eq*).

Pengujian dengan *if/elif/else* dapat menjadi sangat ‘bertele-tele’ dan rumit. Maka dari itu, *case statement* memberikan cara berbeda untuk membuat beberapa pengujian lebih mudah.

```
#!/bin/bash

case "$1" in
hello|hi)
    echo "hello yourself"
    ;;
goodbye)
    echo "nice to have met you"
    echo "I hope to see you again"
```

```
;;
*)
    echo "I didn't understand that"
esac
```

Statement `case` dimulai dengan deskripsi dari ekspresi yang sedang diuji: `case EXPRESSION in`. Ekspresi ini dikutip menjadi `$1`.

Selanjutnya, setiap rangkaian pengujian dijalankan dengan cara mencocokkan pola yang diakhiri dengan tanda kurung tutup. Pada contoh sebelumnya, mencari `hello` atau `hi` ; beberapa *option* dipisahkan oleh tanda vertikal `|` yang merupakan operator OR dalam banyak bahasa pemrograman. Dibawahnya adalah *command* yang akan dijalankan jika pola mengembalikan nilai `true`, yang diakhiri oleh dua titik koma. Pola tersebut terus berulang.

Pola `*` memiliki arti yang sama dengan `else` karena cocok dengan apa pun. Statement `case` mirip dengan statement `if/elif/else` yang dimana pada *statement* itu berhenti apabila kondisi pertama bernilai `true`. Jika tidak ada *option* lain yang cocok, `*` memastikan bahwa yang terakhir akan cocok.

2.4 STRUKTUR PERULANGAN PADA LINUX

Loops memungkinkan kode untuk dijalankan berkali-kali. *Loops* dapat digunakan misalnya ketika kita ingin menjalankan perintah yang sama dari setiap *file* di dalam direktori, atau ketika kita ingin mengulang suatu tindakan hingga 100 kali. Terdapat dua jenis *loops* pada *shell script*, yaitu ***for loop*** dan ***while loop***.

For loop digunakan ketika kita memiliki daftar terbatas yang ingin kita ulangi, seperti *list file*, atau *list nama server*:

```
#!/bin/bash

SERVERS="servera serverb serverc"
for S in $SERVERS; do
    echo "Doing something to $S"
```

```
done
```

Pada script diatas ditetapkan sebuah variabel yang berisi list server yang dipisahkan dengan spasi. Statement **for** kemudian loops dari nama server, setiap kali menetapkan variabel S ke nama server saat ini. Tapi perhatikan bahwa pada S tidak ada tanda dollar, sedangkan pada \$SERVERS ada. Hal tersebut menunjukan bahwa \$SERVERS akan dilanjutkan ke list server. List tidak harus berupa variabel. Berikut dua contoh lain untuk pass sebuah list.

```
#!/bin/bash

for NAME in Sean Jon Isaac David; do
    echo "Hello $NAME"
done

for S in *; do
    echo "Doing something to $S"
done
```

Contoh *loop* pertama berjalan layaknya pada contoh sebelumnya, namun pada contoh ini *list* di pass secara langsung ke *for loop* tanpa menggunakan variabel. Penggunaan variabel dapat membantu dalam hal kejelasan *script* dan seseorang juga dapat lebih mudah membuat perubahan pada variabel dibanding dengan merubahnya pada *loop*.

Contoh *loop* kedua menggunakan "*" yang mana disebut dengan *file glob*. Yang merupakan kumpulan *file file* pada direktori saat ini / *current directory*.

Jenis *loop* lainnya adalah **while loop**, yang beroperasi pada *list* yang perulangannya tidak pasti tapi bergantung pada suatu kondisi yang harus dipenuhi. *While loop* terus berjalan dan pada setiap itersi menjalankan perintah *test* untuk melihat apakah kondisi harus dijalankan lain waktu. Dengan kata lain, kita dapat menyebutnya "**while some condition is true, do stuff**" atau selagi sebuah kondisi bernilai *true*, lakukan sesuatu.

```
#!/bin/bash

i=0
while [ $i -lt 10 ]; do
    echo $i
    i=$(( $i + 1 ))
done
echo "Done counting"
```

Contoh diatas menunjukkan sebuah *while loop* yang menghitung dari angka 1 sampai 9. Variabel penghitung, *i* diberi nilai awal 0. Kemudian perulangan *while* dijalankan dengan perintah *test* seperti “apakah \$i lebih kecil daripada 10?”. Perlu diperhatikan bahwa perulangan *while* menggunakan notasi penulisan yang sama dengan percabangan *if*.

Pada perulangan *while* tersebut nilai sekarang dari *i* dicetak dan selanjutnya 1 ditambahkan ke dalam perintah \$ ((operasi)) dan menetapkan kembali nilai *i* yang baru. Ketika nilai dari *i* sudah mencapai 10, maka *while* akan mengembalikan nilai *false* dan proses *loop* berhenti, kemudian menjalankan proses setelah *loop*.

RANGKUMAN

1. *Shell script* adalah *file* yang berisikan *command* yang dapat di eksekusi yang disimpan dalam bentuk *text file*. Ketika *file* dijalankan, setiap *command* akan di eksekusi.
2. *Shell script* biasanya diawali dengan baris *shebang* yang menuju ke *path* interpreter, seperti *bash*, *sh*, dll yang akan memanggil interpreter tersebut sebagai interpreter *file* ketika *script* dijalankan.
3. Variabel adalah bagian penting dari bahasa pemrograman yang berguna sebagai tempat menyimpan nilai sementara pada *script*.
4. Percabangan *if* digunakan untuk membandingkan suatu kondisi dan menjalankan *statement* apabila kondisi tersebut bernilai *true*. Formatnya:

```
if [ kondisi_1 ]; then
    perintah_1
elif [ kondisi_2 ]; then
    perintah_2
.
.
else
    perintah_3
fi
```

5. Percabangan *case* digunakan digunakan untuk membuat beberapa pengujian menjadi lebih mudah. Dengan menggunakan *case*, kondisi dapat dikelompokkan secara logis dan lebih mudah dan jelas dalam penulisannya.

Formatnya:

```
case $variabel in
    isi_1)
        perintah_1
        ...
;;
    isi_2)
        perintah_2
        ...
;;
    *)
        perintah_3
        ...
esac
```

6. Perulangan *for* digunakan untuk melakukan pekerjaan berulang sebanyak daftar atau *list* yang disediakan. Formatnya:

```
for variabel in (daftar argumen); do
    perintah
    ...
done
```

7. Perulangan *while* beroperasi pada *list* yang jumlah perulangannya tidak pasti, dan bergantung pada suatu kondisi yang harus dipenuhi. Formatnya:

```
while [ kondisi ]; do
    perintah
    ...
done
```